

浙江大学CAD&CG 国家重点实验室可视化与分析小组

支持多终端的可视化平台设计

陈海东
2013/4/25

更新记录

时间	更改者	概述
2013 年 4 月 25 日星期四	陈海东	配置设计文档
2013 年 5 月 10 日星期五	陈海东	绘制系统设计文档编写
2013 年 5 月 12 日星期日	陈海东	内存管理模块设计，纹理资源管理模块设计，着色器程序管理模块
2013 年 5 月 13 日星期一	陈海东	UI 系统设计，插件系统设计
2013 年 5 月 14 日星期二	陈海东	网络系统设计，数学系统，脚本系统，编程规范

目录

第一章 总览.....1

 1.1 概述1

 1.2 子系统简介.....2

第二章 内存管理.....3

第三章 绘制系统.....4

 3.1 事件管理.....5

 3.2 绘制参数管理.....5

 3.3 绘制场景管理.....6

 3.4 消息管理.....7

 3.5 纹理资源管理.....8

 3.6 着色器程序管理.....11

 3.7 构建绘制场景.....12

第四章 用户界面系统.....13

第五章 插件系统.....16

第六章 网络系统.....17

 6.1 网络系统应用场景.....17

 6.2 网络系统静态结构设计.....18

第七章 数据系统.....19

第八章 数学系统.....21

第九章 脚本系统.....21

附录 A: 编程规范22

 文件格式规范.....22

 作用域规范.....22

 类相关规范.....24

 命名规范.....26

 注释规范.....27

 格式规范.....29

第一章 总览

1.1 概述

VisNG 是一款由浙江大学可视化分析小组自主研发的并行可视化及分析引擎。该引擎支持 C/S 和 B/S 模式。VisNG 支持超大分辨率屏幕显示。VisNG 既可以部署于普通 PC 机，作为客户端程序运行于本机，亦可运行于并行可视化系统的节点机器，为基于浏览器和大屏可视化系统提供基本的可视化服务。

概括地讲，VisNG 包括以下子系统：Render System, UI System, Plugin System, Network System, Data System, Script System。

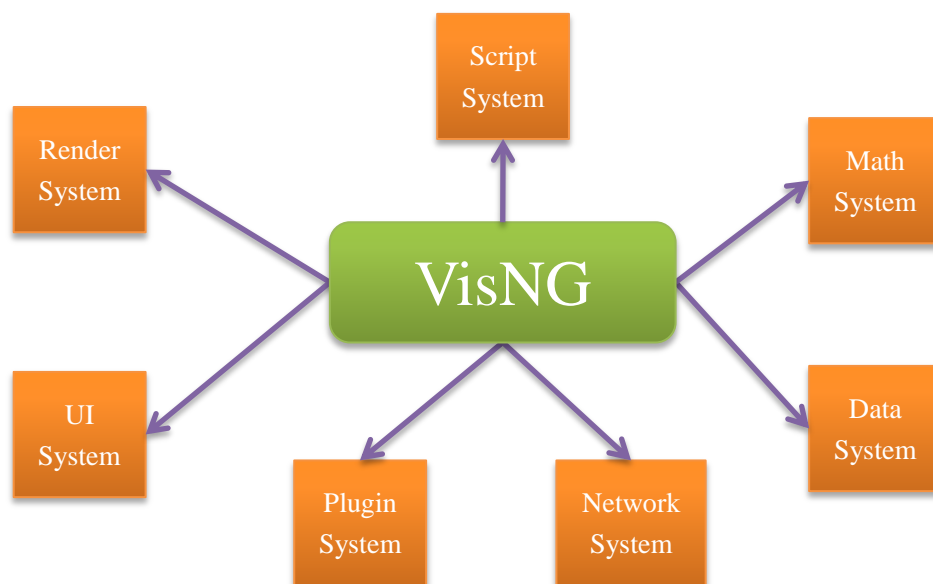


图 1 VisNG 系统示意图

基于浏览器和大屏的可视化系统要求具有强大的后台计算集群作提供基本的可视化服务。而浏览器和大屏幕作为交互和显示终端。Appache 服务器负责解析浏览器和大屏可视化系统发出的可视化任务，然后将任务交于部署了 VisNG 的 CPU/GPU 集群完成可视化，并返回相应的可视化结果。可视化期间所需要的数据由数据库提供，通过高性能网关进行访问。



图 2 并行架构示意图

1.2 子系统简介

- ◆ **Render System** 绘制系统主要负责：场景的管理（可视化对象的管理），对象的绘制。
- ◆ **UI System** 交互界面系统主要负责：界面元素的创建和布局，消息的转发及响应。
- ◆ **Plugin System** 插件系统主要负责：主要负责系统插件和用户插件的加载及管理。
- ◆ **Network System** 网络系统主要负责：以线程为基本单位对 Http 通信、socket 通信进行管理。
- ◆ **Data System** 数据系统主要负责：数据的请求、加载、卸载，支持数据库或文件系统两种数据访问模式。
- ◆ **Math System** 数学系统主要负责：数学运算，物理模拟。
- ◆ **Script System** 脚本系统主要负责：脚本解析配置可视化引擎。

第二章 内存管理

由于复杂的引擎逻辑，一个对象常常会被多个不同的对象所引用，将对象的内存管理完全交给用户控制是一件十分困难的事情，因为用户需要明确知道对象的生命周期和对象之间的相互关系。因此，本引擎采用引用计数策略实现引擎对象的自动化内存管理。

所有引擎对象必须继承自 **IRCBASE**。该类实现了基本的引用计数功能。为了保证可完全监测引擎的内存使用情况，要求调用者只能通过 **ObjectFactory** 提供的接口实例化和销毁某个具体对象。当类被实例化为对象后，其默认的 `referenceCounter_` 将被置为 1。至此，任何指针在指向该对象之前必须先调用 `retain` 函数增加引用计数。不需要该指针时则需要调用 `release` 函数递减引用计数。当对象的引用计数为 0 时，则自动调用析构函数释放其申请的内存。函数 `retain` 和 `release` 必须成对使用，否则会造成内存泄露。值得注意的是，仅当用户需要长期使用该对象的指针，且不知道何时释放该对象时，才需要调用引用计数函数，而临时指针不需要调用引用计数相关函数。对于分配在栈上的对象，为避免错误的内存释放请求，请不要调用 `retain` 和 `release` 函数。如下例所示。

```
Class A : public IRCBase
{
    public:
        A();
        ~A();
        void doSomething();
}

void TestFunctionHeap()
{
    A *a = SAFE_NEW("A");    // referenceCounter_ is 1
    a->retain();
    A *b = a;                // referenceCounter_ is 2
    A *c = a;                // referenceCounter_ is 2
    c->doSomethign();         // Temporary pointer does not have to call retain and release.
    b->release();              // referenceCounter_ is 1
    SAFE_DELETE(a)           // referenceCounter_ is 0. Destructor will be called automatically.
}

void TestFunctionStack()
{
    A a;
    a.retain();              // This is not allowed.
    a.release();              // This is not allowed.
}
```

注意：目前该引擎系统无法处理循环引用，请用户使用时注意指针的赋值顺序。

为了支持脚本语言开辟引擎对象，本引擎采用反射机制实现对象的注册、分配、类型转换和销毁。所有开放给脚本语言的引擎对象必须使用宏 **REGISTER_REFLECTION_CLASS** 进行注册。注册过程实则是将类名和对象分配函数指针记录与 **ObjectFactory** 的单例对象中。任何

用户可通过 **ObjectFactory** 的 `createObject` 函数返回一个给定类名的对象，通过 `deleteObject` 释放给定对象名的引擎对象。

引擎中的资源对象（如纹理、着色器程序等）常常需要被多次创建和访问。然而，这些资源的创建需要许多 IO 操作，为了提高效率，引擎引入缓存机制解决资源频繁访问的需求。所有可缓存资源对象必须实现 **ICacheableResource** 接口，**CacheManager** 使用字符串作为资源管理的键值。

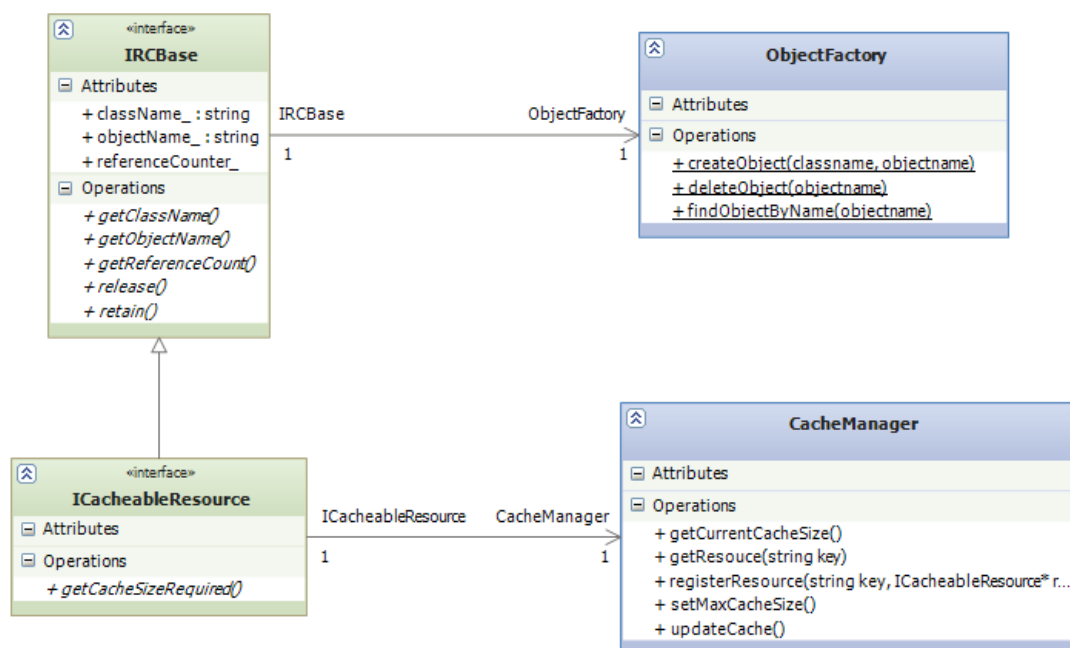


图 3 内存管理基本类

第三章 绘制系统

绘制系统主要包含：绘制参数的管理，绘制对象的管理，绘制设备事件的管理、引擎内部消息的管理、纹理的管理、着色器的管理等等。

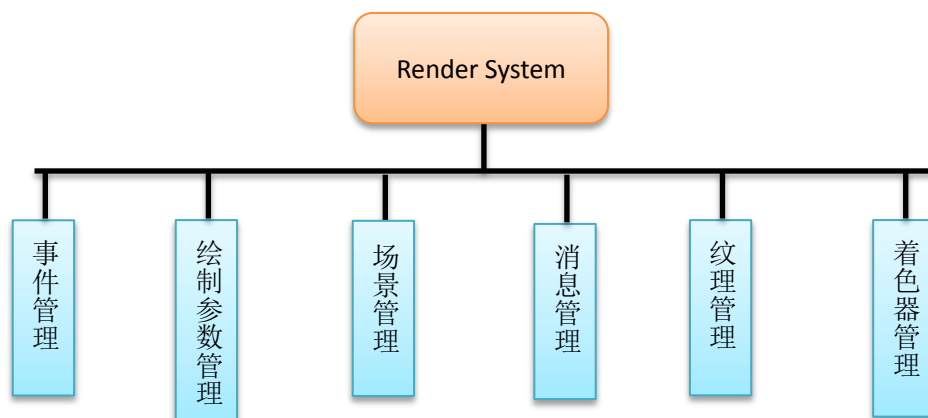


图 4 绘制系统模块图

3.1 事件管理

EventHandler 主要用于处理绘制设备发出的事件。每个绘制设备拥有一个 **EventHandler**。每个 **EventHandler** 可以添加多个 **EventListener** 用于监听并响应绘制设备产生的事件。目前，引擎支持三大类事件：**MouseEvent**、**KeyEvent** 和 **TimeEvent**。

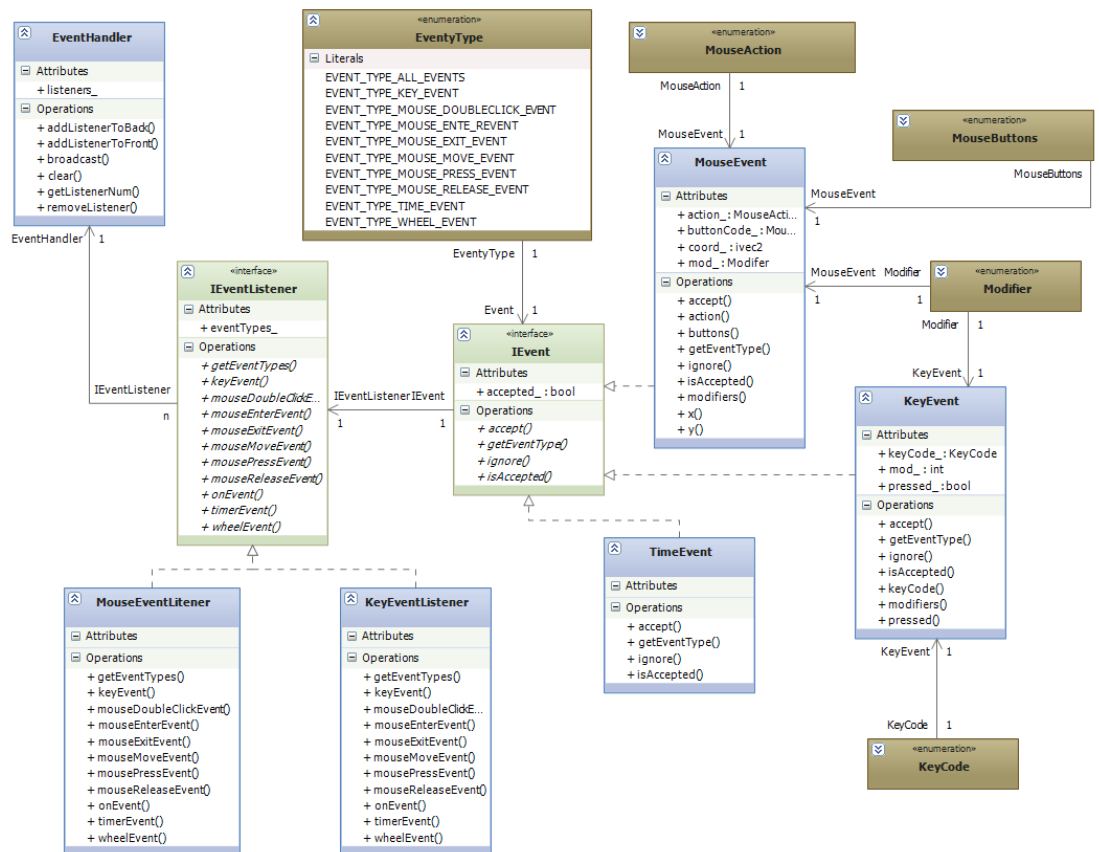


图 5 事件响应模块类图

3.2 绘制参数管理

目前，引擎提供一些基本的相机操作类如：UVN 相机，欧拉相机，FPS 相机和地球相机。地球相机主要用于以地球为基本可视化信息载体的应用情形。由于数据更新需要长时间的网络等待，为了保证交互的流畅性，相机在变动过程中引擎不会发出数据更新请求，而是当交互停止的瞬间由相机操作器（也就是类 **XXCameraNavigator**）发出 **MSG_UPDATE_DATASET** 消息，任何实现了 **IMessageReceiver** 接口并注册 **MSG_UPDATE_DATASET** 的对象都将响应该消息更新可视化所需的数据集。

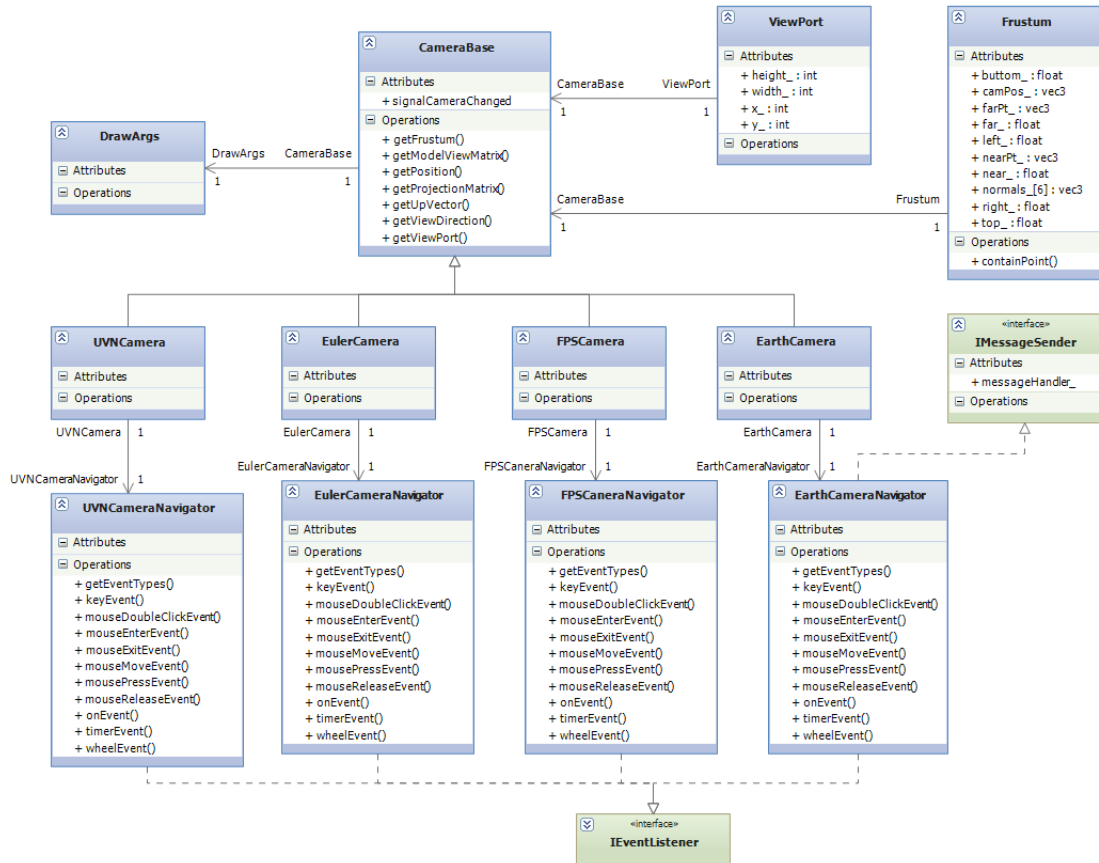


图 6 绘制参数模块类图

3.3 绘制场景管理

所有场景对象都继承自同一接口 **ISceneObject**。所有场景对象被分为：可绘制对象和不可绘制对象。**RenderableSceneObj** 主要负责调用 OpenGL 代码实现绘制。而 **NonRenderableObj** 是所有不可绘制对象的基类，主要负责逻辑控制和管理，如 **RenderableObjGroup** 类用于将多个可绘制对象组织在一起。每个场景对象可通过 **sleep** 和 **awake** 函数改变场景对象的状态。如果场景对象被设置为睡眠状态，绘制每一帧时该对象将不会被更新，否则将会调用 **update** 函数更新该对象。

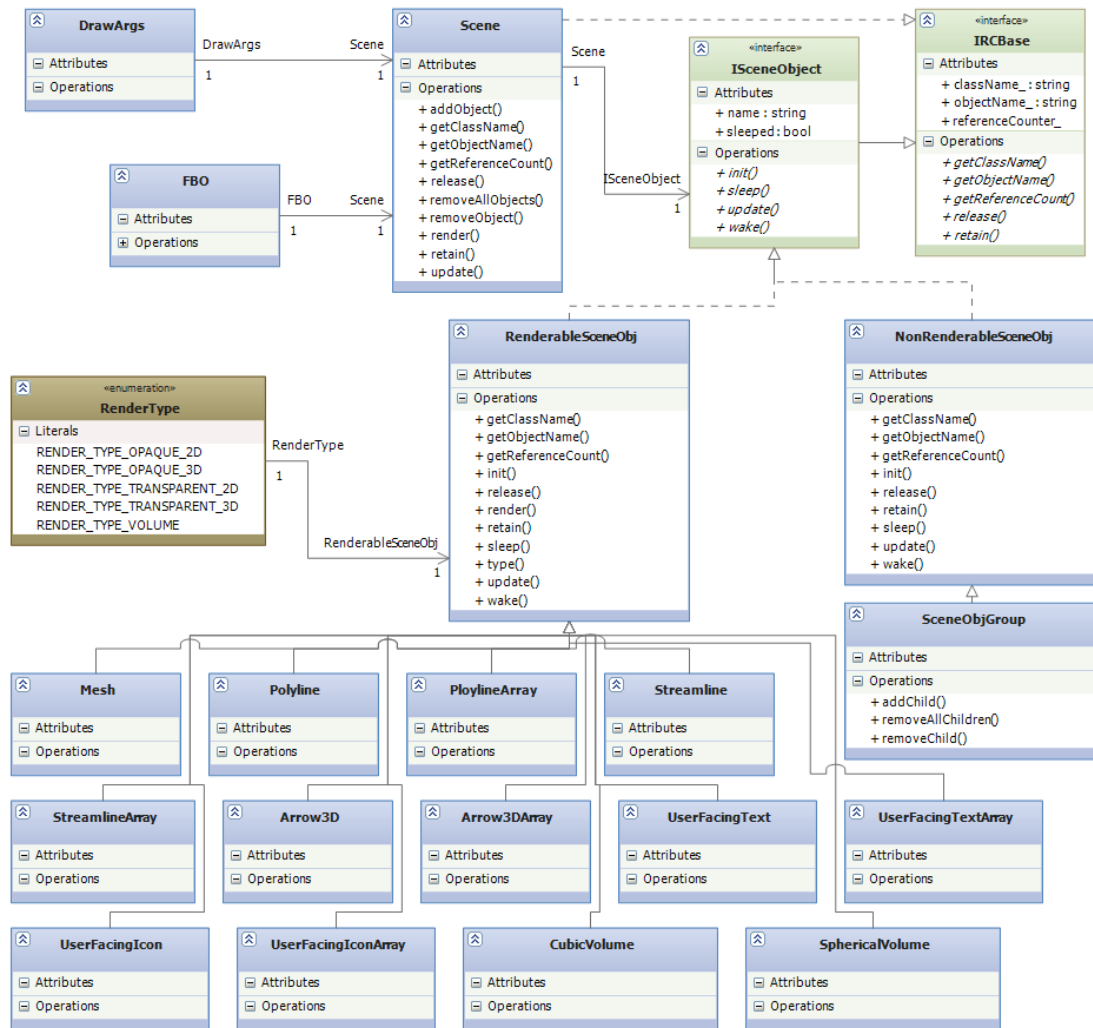
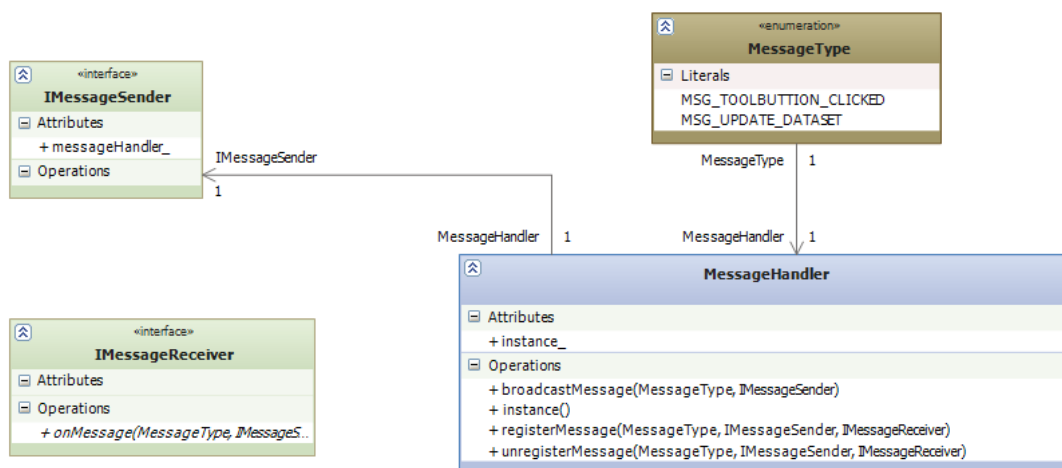


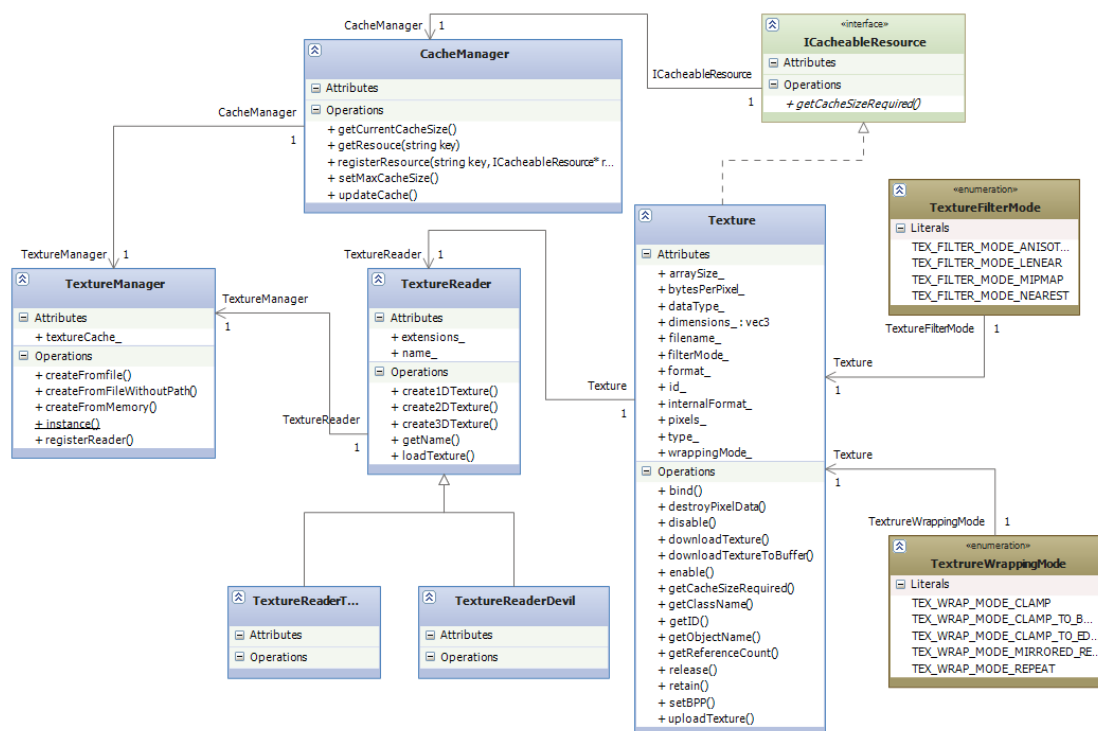
图 7 绘制场景管理模块类图

3.4 消息管理

消息响应模块主要用于注册，转发用户定义的消息。可发出消息的类必须实现接口 **IMessageSender**。用于响应消息的类必须实现接口 **IMessageReceiver**。**MessageHandler** 主要负责注册、转发消息等。



3.5 纹理资源管理



任何图像文件都可作为纹理的，为了实现扩展性，引擎通过注册 **TextureReader** 实现支持类型的纹理文件的读取。当用户需要一个纹理资源时，**TextureManager** 首先会在 **Cache** 中查找该纹理，如果该纹理已存在与显存中，则直接返回该纹理对象于用户；如果不存在，则遍历所有的已注册的纹理读取器，选择扩展名匹配的纹理读取器，并调用该读取器从文件中加载纹理数据，并开辟显存资源，将纹理数据拷贝至显卡，最后将生成的纹理在 **Cache**

中注册一遍下次快速访问。如果 **TextureManager** 返回的纹理不为空，表示创建资源成功。使用示例代码如下：

```
void TextureTest()
{
    // Loading texture. Reference counter will be incremented automatically.
    Texture* tex = TextureManager::instance()->load("test.png");
    if (0 == tex) {
        return;
    }
    tex.retain();
    ...    // do something else with this texture
    tex.release();
    tex = 0;
}
```

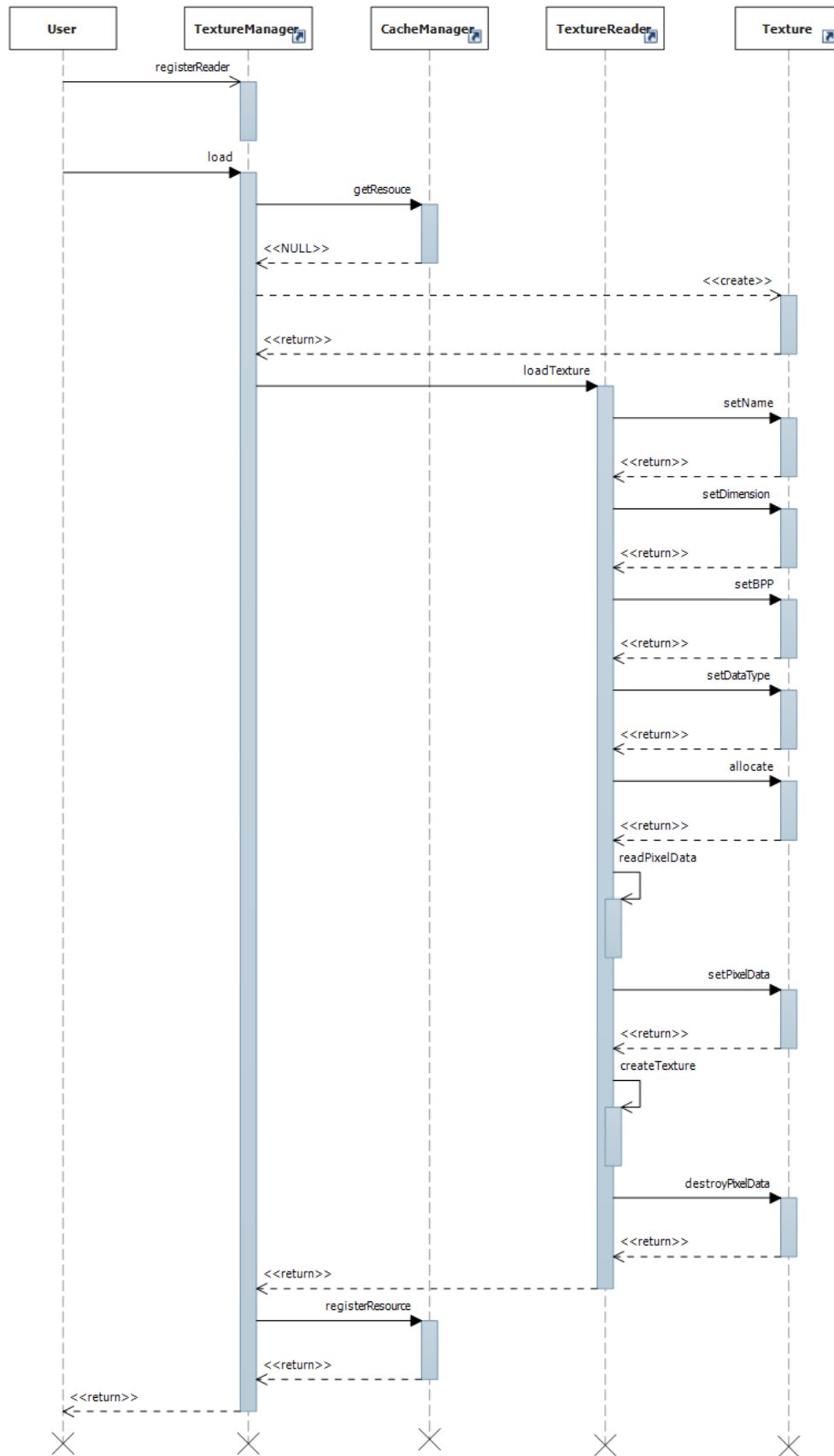


图 10 纹理资源管理模块运行机制

3.6 着色器程序管理

引擎仅支持基于 GLSL 的着色器。与纹理资源管理模块类似，该模块也采用着色器文件名的组合（如：“test.vert#test.frag#”）作为 Cache 的关键值以保证相同着色器程序在内存中只有一份拷贝。

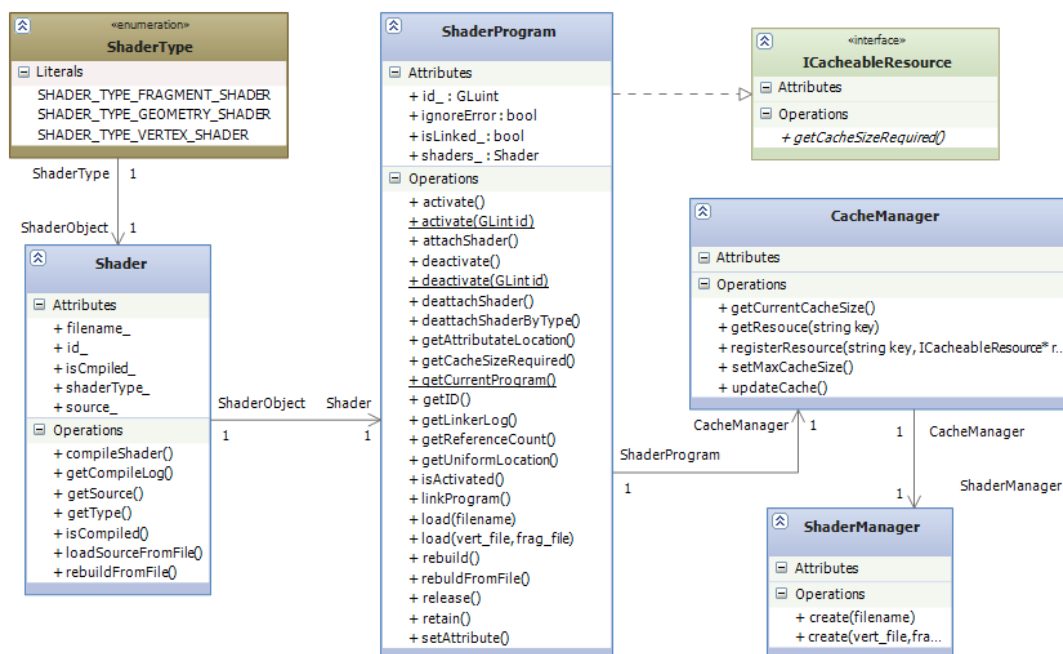


图 11 着色器程序管理模块

与加载纹理资源类似，着色器程序的管理也采用基于缓存的机制降低重复访问和使用的代价。**ShaderProgram** 包含可绑定多个着色器实例。**Shader** 对应具体的着色器，如顶点着色器、片段着色器、几何着色器等。

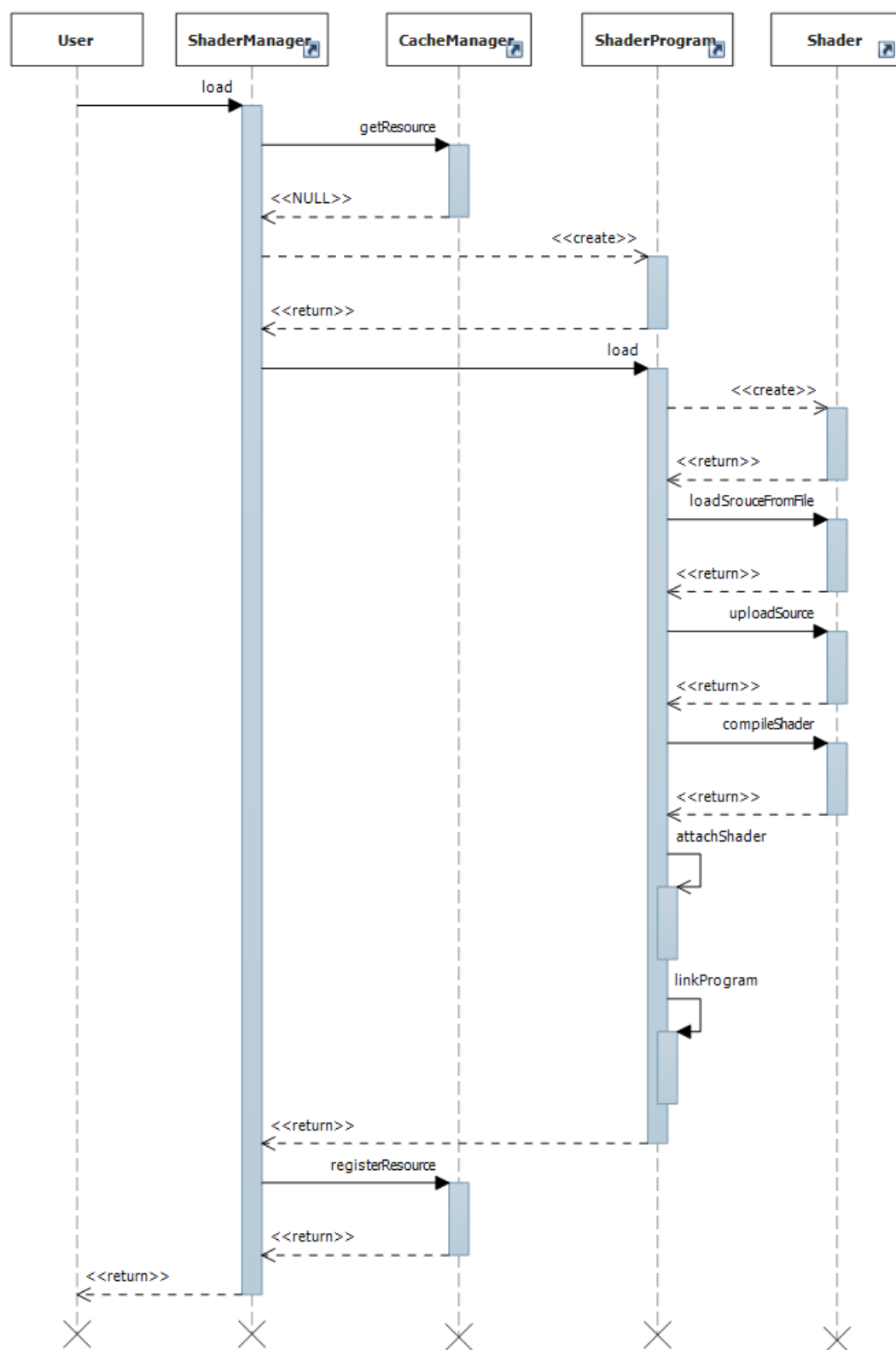


图 12 加载着色器程序

3.7 构建绘制场景

构建绘制场景时，用户先构建场景管理对象，然后创建绘制参数类，并将其绑定至场景

管理对象。接着逐步创建相应的场景对象，并添加至场景管理器中。待添加完所有对象后，需要调用场景管理器的排序函数，对所有场景对象进行排序。**Sort** 函数首先将所有可绘制对象根据不同的绘制类型进行分类，对同一类的所有绘制对象则根据优先级进行排序。具体细节可参考混合可视化框架设计文档。

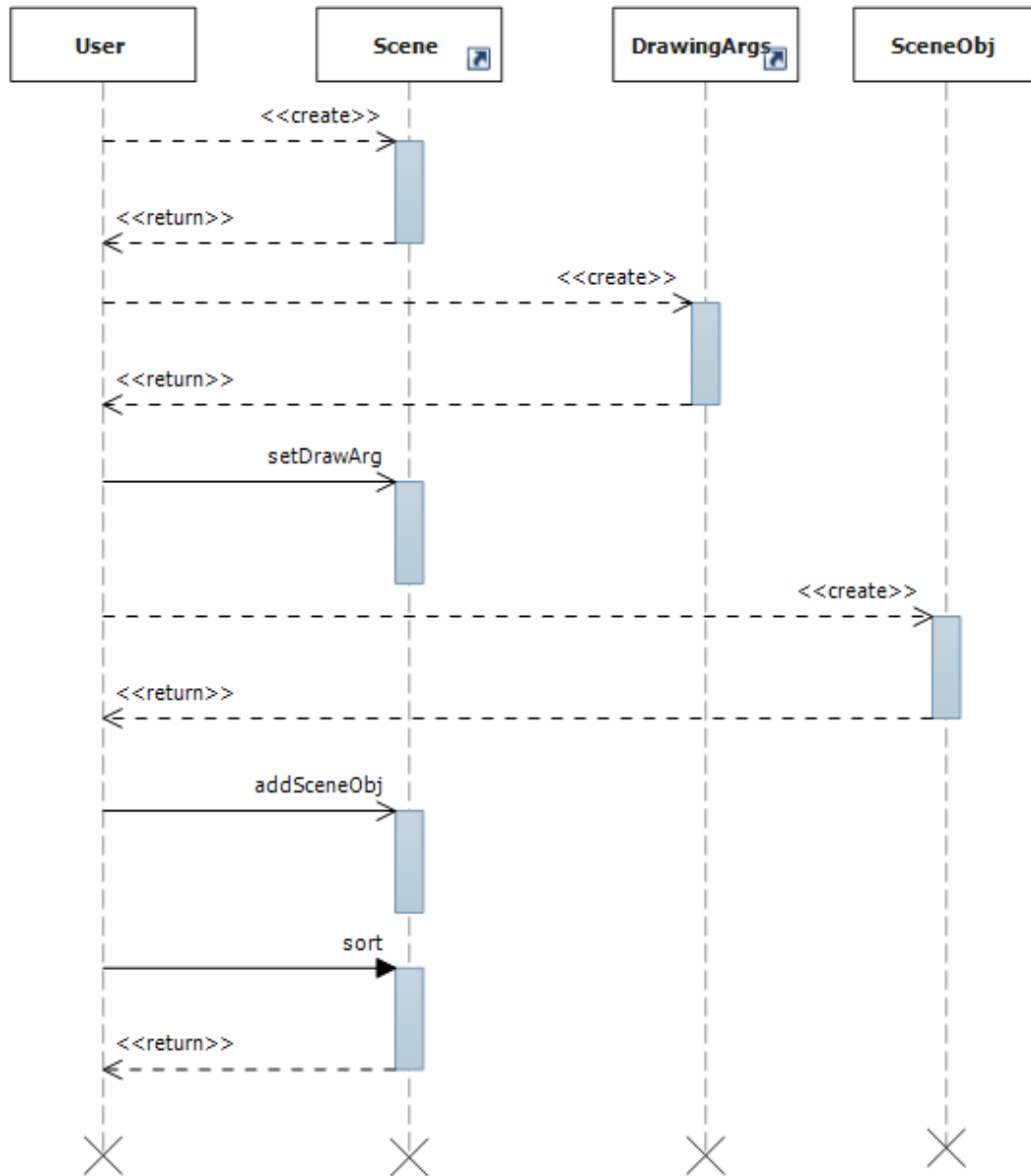


图 13 构建绘制场景过程

第四章 用户界面系统

引擎的 GUI 元素主要由两大部分组成：基于 Qt 的 UI 部件和引擎本身提供的 UI 元素。基于 Qt 的用户界面部件的绘制由 Qt 进行管理，而引擎提供的 UI 元素的则由 **GUIManager**

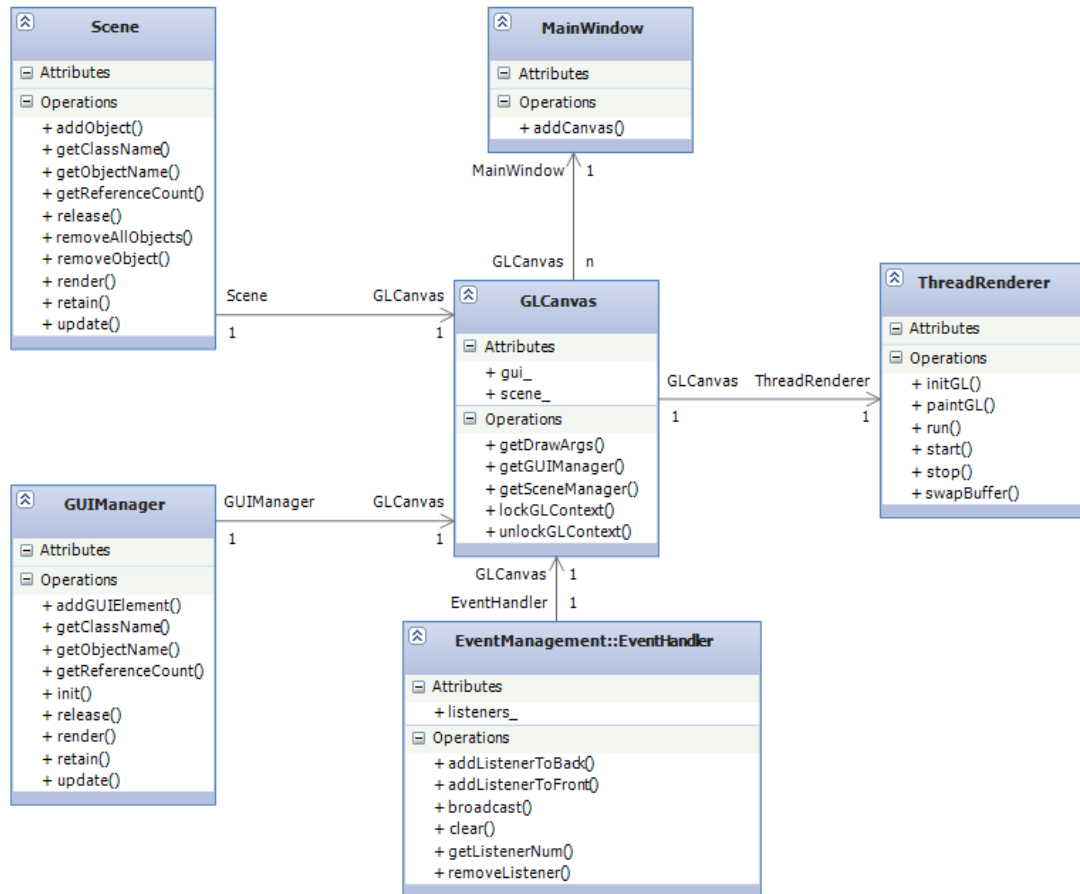


图 15 基于 Qt 的 UI 系统结构图

当用户开始执行绘制线程时，首先需要获取绘制设备的互斥锁，然后初始化 OpenGL 配置，接着更新和绘制场景，然后更新和绘制交互界面元素，最后交换缓存并释放获取的互斥锁，具体流程如下图所示。

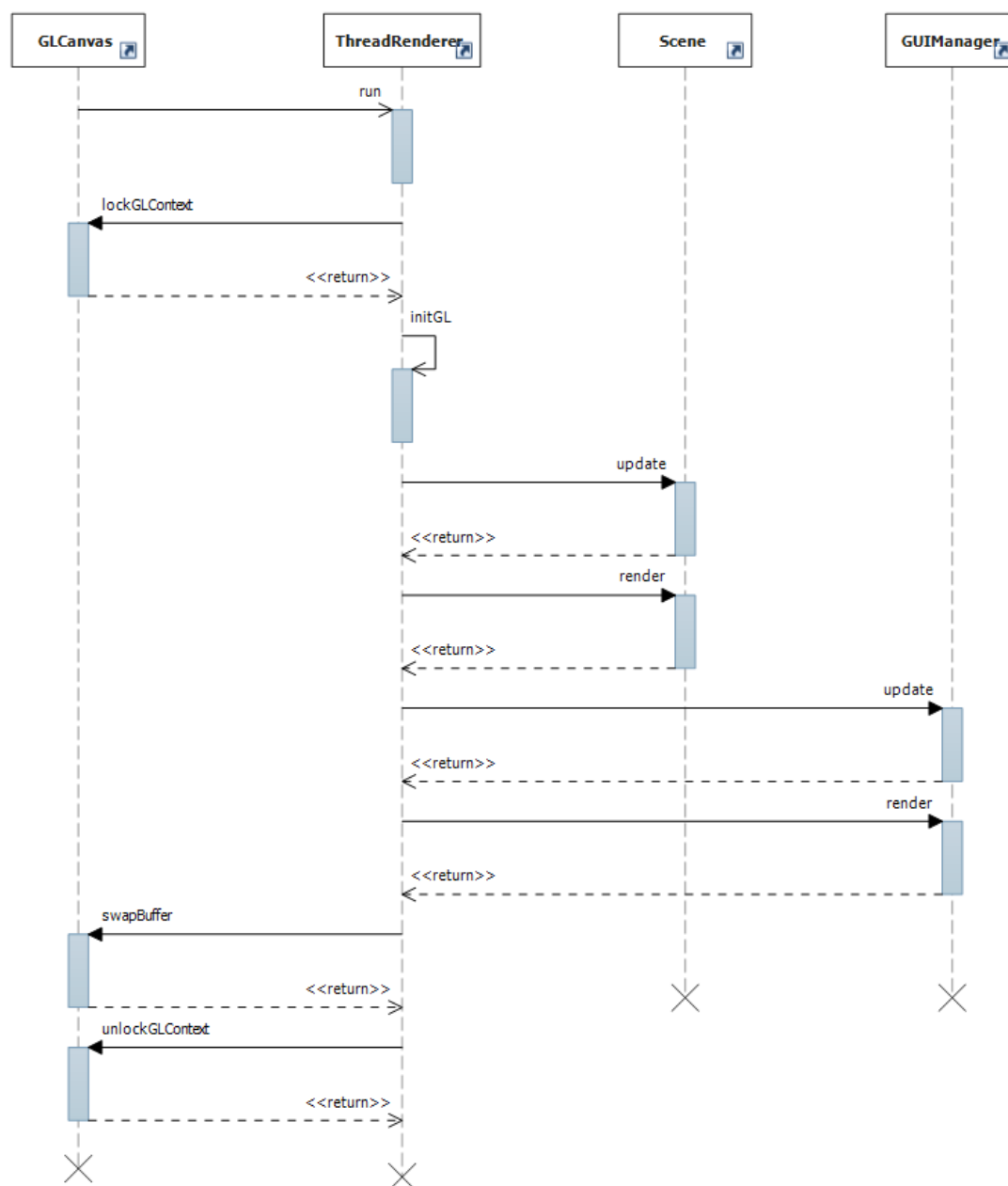


图 16 绘制线程的运行机制

第五章 插件系统

插件主要用于扩展引擎的功能，以便第三方扩展引擎的基本功能实现二次开发。所有插件需实现接口 *IPlugin*。*PluginManager* 负责从动态库中加载插件，读取其中的基本信息，并完成版本兼容性验证，最后初始化插件。

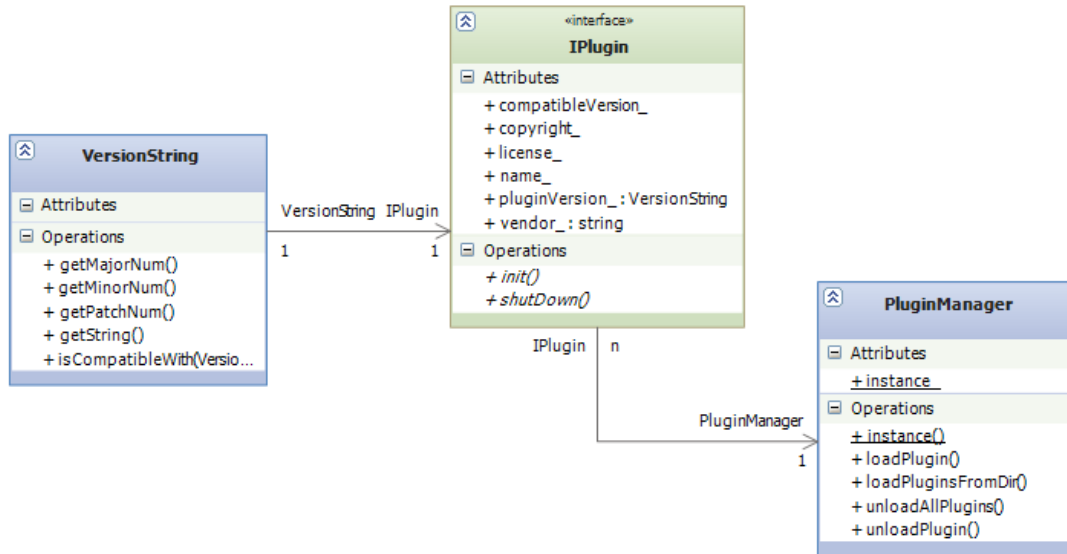


图 17 插件系统类图

第六章 网络系统

6.1 网络系统应用场景

网络系统主要为两大用户提供基础的网络通讯服务：引擎（VisNG）和用户（如：基于浏览器的可视化系统，大屏可视化系统的 Master 节点）。

引擎主要通过网络系统向服务器提出资源下载请求。目前，系统提供两种资源下载方式，包括基于 **Http** 协议的资源下载和基于 **Socket** 通信的资源下载模式。

而用户则可以通过引擎的网络系统与引擎进行通信，进而控制引擎的运行。具体包括：用户可通过网络系统上传脚本配置基本的绘制环境；当绘制参数发生改变时，用户可通过发送消息至引擎的网络系统实现对绘制环境的更新等。

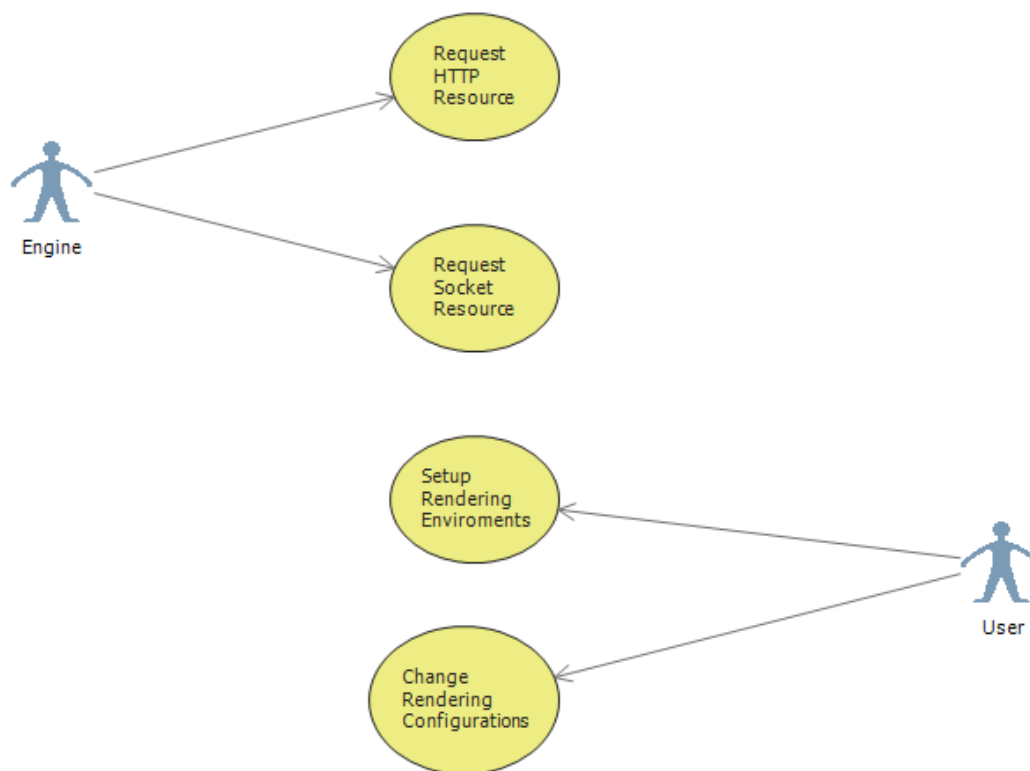


图 18 网络系统应用案例

6.2 网络系统静态结构设计

NetworkManager 是一个单例类，用于向引擎的其它子系统提供统一的访问接口。**HttpDownloader** 类主要负责下载基于 Http 协议的资源。**TCPSocketDownloader** 则主要用于建立 TCP 连接并下载资源。**CommandListener** 用于监听和解析用户发出的各种引擎操作命令（如：创建新的绘制流程，更新某个绘制流程的绘制参数等），然后将这些操作命令组织成脚本，由脚本系统负责调度引擎完成用户的请求。

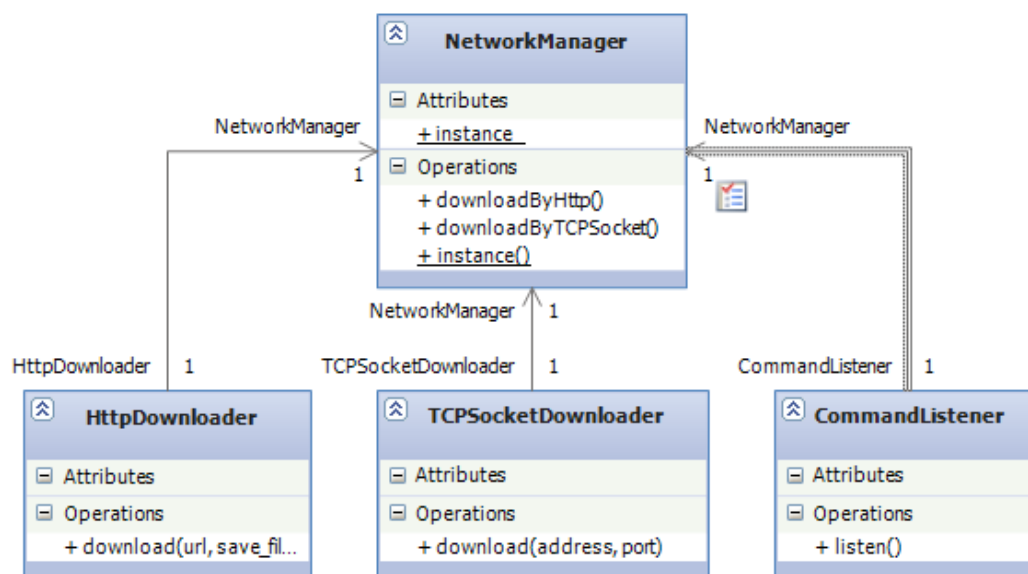


图 19 网络系统类图

第七章 数据系统

数据系统提供引擎支持的所有基本数据类型。所有数据对象类均继承自 *IDataObject*。为了支持数据对象可在多个线程或对象之间共享，采用了数据池的技术。任何放入数据池的对象均可被其它对象所访问，从数据池中获取对象时需要指定关键值。

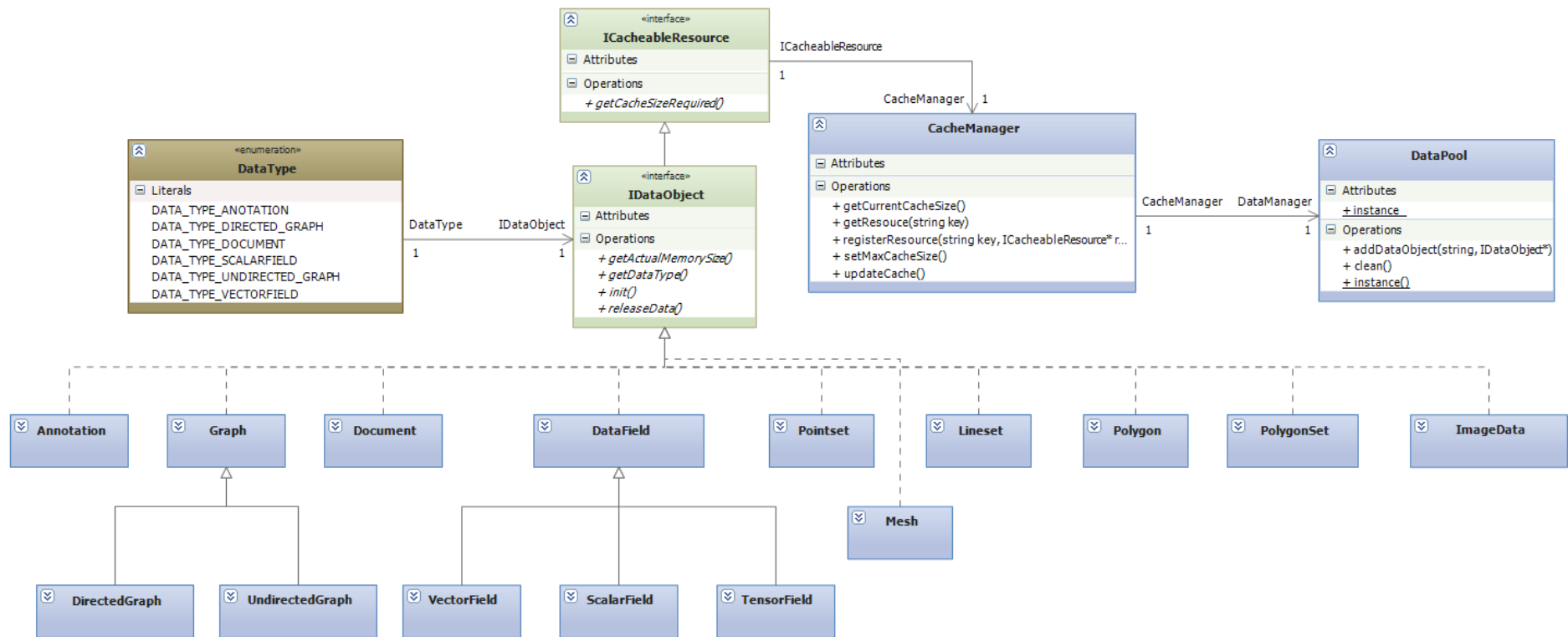


图 20 数据系统类图

第八章 数学系统

目前，引擎所有的数学和物理计算均采用第三方库。向量、矩阵和四元组的计算采用 **CML** (<http://cmldev.net/>) 库。为了语法简便性和可读性，需要使用 **typedef** 对已有类型和常量等进行重新命名。

第九章 脚本系统

引擎采用 **AngleScript** (<http://www.angelcode.com/angelscript/>) 作为可视化脚本语言的解析和执行引擎。

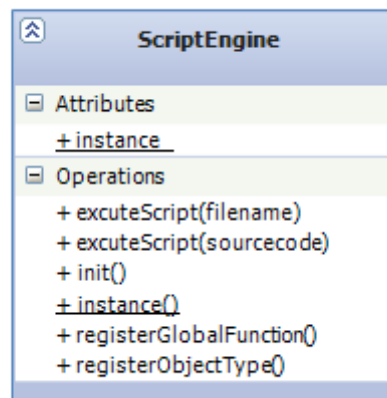


图 21 脚本解析和执行引擎类图

附录 A：编程规范

本编程规范主要参考自 Google 公司采用的 C++ 编程要求，如有不明白之处可参考其在线文档（<http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml>）。

文件格式规范

采用小写命名文件名

所有文件必须采用小写格式，允许采用下划线连接不同的单词，尽量避免缩写。

使用 `#define` 避免重复引用

所有头文件必须采用 `#define` 避免该文件被多系包含。符号名必须遵从 `<PROJECT>_<PATH>_<FILE>_H_`。如“src/math/vector.h”需要采用以下宏定义：

```
#ifndef VISNG_MATH_VECTOR_H_
#define VISNG_MATH_VECTOR_H_
...
#endif
```

尽可能使用向前声明

为了避免过多的无必要的 `#include`（注：在头文件中包含过多的其他头文件会降低编译效率，增加编译时间），请尽量使用向前声明。

函数参数顺序：先输入参数然后输出参数

声明函数时，保证先输入参数，然后是输入参数，最后是带默认值的参数。

`#include` 顺序：先用户定义文件，然后系统库文件

在头文件和源文件中保证用户定义文件先被包含，然后引用系统库文件。

作用域规范

命名空间规范

所有程序代码必须位于一个给定的命名空间下，命名空间名字要求不超过 8 个字符。头文件和源文件需要显示的使用命名空间。使用时必须通过命名空间访问程序代码。如下例所示：

```
// in the header file
namespace myspace {
    class TestClass {
    public:
        void do();
    }
```

```

    }
}

// in the source file
namespace myspace {
    TestClass::do() {
        ...
    }
}

// for use
// using namespace myspace; // This is not allowed.
myspace:: TestClass *a = new myspace:: TestClass();

```

静态变量和函数

尽量将全局静态函数和变量定义域命名空间中，使用时通过命名空间进行访问。如下例代码所示：

```

namespace myspace {
    float PI = 3.14f;
    void testFunc() {
        ...
    }
}

printf("%f\n", myspace::PI);

```

尽量避免使用静态全局变量（除非是程序常量值）。因为静态全局变量非常不易于程序调试。

局部变量定义时便进行初始化

为了保证变量都被初始化，尽量保证定义变量时便对其进行初始化。如下例所示：

```

int cnt;
cnt = 0;    // Bad – declaration and initialization are separated.

int cnt = 0;    // Good

```

尽可能避免临时对象的生成

由于变量作用域的不同，常常会生成很多临时对象，这些临时对象会增加程序的开销，因此鼓励使用引用或者指针作为函数参数，尽可能延长变量的作用域。如下例所示：

```

void testFunc(TestClass a) {    // This is not a very good option, a temp object will be created
    ...
}

```

```
void testFunc(const TestClass& a) { // Good
    ...
}
```

类相关规范

构造函数规范

尽量避免将复杂的函数逻辑置于构造函数中。鼓励使用初始化列表初始化类的所有成员变量。初始化列表的顺序必须与编译器的初始化顺序保持一致。如下例所示：

```
// in the header file
class TestClass : public BaseClass {
public:
    TestClass();
    virtual ~TestClass();

    int count_;
    float length_;
}

// in the source file
TestClass::TestClass()
    : BaseClass()
    , count_(0) , length_(1.0f) {
    ...
}
TestClass::~~ TestClass() {
    ...
}
```

如果没有其他构造函数，必须显式地提供一个默认构造函数，尽量避免让编译器提供默认构造函数。如下例所示：

```
class TestClass { // No constructor exist. It is forbidden.
public:
    void do();
}

class TestClass {
public:
    TestClass();
    ~TestClass();
    void do();
}
```

除非确实需要拷贝构造函数和赋值操作符，否则请调用 **DISALLOW_COPY_AND_ASSIGN** 禁止编译器提供默认拷贝构造函数和赋值操作符。如下例所示：

```
// A macro to disallow the copy constructor and operator= functions
// This should be used in the private: declarations for a class
#define DISALLOW_COPY_AND_ASSIGN(ClassName) \
    ClassName(const ClassName&) {}           \
    void operator=(const ClassName&) {}

// used in class TestClass
class TestClass {
public:
    TestClass();
    ~TestClass();

private:
    DISALLOW_COPY_AND_ASSIGN(TestClass)
}
```

保证所有的子类都采用 **public** 继承方式。父类的析构函数必须是虚函数，也就有修饰符 **virtual**。如果子类需要访问父类的成员变量，请慎重使用 **protected** 访问修饰符。

在类的声明文件中保证，**public** 的成员函数在最前面，**protected** 函数次之，**protected** 和 **private** 成员变量或函数最后。为了代码的可读性，请保证函数在声明文件中出现的顺序和源文件中出现的顺序保持一致。如下例所示：

```
class TestClass {
public:
    TestClass();
    virtual ~ TestClass();

    void perform();

protected:
    void run();

private:
    int num_;
}
```

命名规范

总体命名规则

所有变量名，函数名，类名等必须具有实际意义，避免使用一些无意义的名字和晦涩的缩写词。如下例所示：

```
int num;    // Good
int n;      // Bad

int error_count; // Good
int err_cnt;    // Bad
```

类型命名

命名类、结构体、枚举等时必须保证单词首字母大写，禁止使用下划线作为连词符。如下例所示：

```
class HttpRequest { ...
struct UserInfomation { ...

typedef std::vector<int> UserID;

enum ObjectType { ...
```

变量命名

所有局部变量都使用小写，并采用下滑线作为单词连接符。成员变量必须在变量的结尾再添加一个下划线。如下例所示：

```
int user_count;    // local variable

class TestClass {
    int user_count_; // member variable
}

struct UserInfomation {
    int name;
    int user_id;
}

void testFunc(int num_errors) { ... }
```

常量命名

所有常量名均采用大写，使用下划线作为连词符。为了与宏进行区分，常量名的首字母必须是 **K**。如下例所示：

```
const int K_DAYS_IN_A_WEEK = 7;
```

函数命名

函数名均采用动宾结构或者只保留动词，不允许将名词作为函数名。动词的首字采用小写。所有私有成员变量可通过 **setters** 和 **getters** 进行访问。如下例所示：

```
void sendRequest() { ... }

class Player {
public:
    Player() {}
    ~ Player(){}
    void goForward();
    void shoot();
    int getAge() { return age_; }

private:
    int age_;
}
```

枚举类型命名

枚举值可以看作是常量。其访问需要紧跟类型名。因此，不需要加 **K** 标示为常量。但需要有类型名（或其缩写）作为前缀。且必须初始化第一个枚举值。如下例所示：

```
enum ObjectType {
    OBJ_TYPE_POINT = 0,
    OBJ_TYPE_LINE,
    OBJ_TYPE_MESH,
};
```

宏命名

宏的所有字母都采用大写。尽量避免用宏定义变量，因为宏不具类型，使用时会进行类型转换丢失精度或产生错误。使用示例如下：

```
#define SAFE_DELETE(p) ...

#define PI 3.14 // Not recommended. Global constant is recommended.
```

注释规范

所有代码保持统一注释风格，// 和 /* */均可。

类注释

每个类的开头需要有注释说明该类的主要功能，必要时可以给出使用样例代码。如所示：

```
// Iterates over the contents of a GargantuanTable. Sample usage:
//     GargantuanTableIterator* iter = table->NewIterator();
//     for (iter->Seek("foo"); !iter->done(); iter->Next()) {
//         process(iter->key(), iter->value());
```

```
//    }
//    delete iter;
class GargantuanTableIterator {
    ...
};
```

函数注释

公共成员函数必须有注释说明其主要功能、参数、和返回值。如下例所示：

```
/*
 * Return the sum of given two int variables.
 *
 * @param v1 The first input parameter.
 * @param v2 The second input parameter.
 *
 * @return The sum of v1 and v2.
 */
int add(int v1, int v2) { return v1 + v2; }
```

变量注释

任何变量都可添加注释。注释要求在出现变量的上一行或多行。如下例所示：

```
// The number of errors
int num_errors;
string name;
```

代码体注释

多行注释

```
// Divide result by two, taking into account that x
// contains the carry from the add.
for (int i = 0; i < result->size(); i++) {
    x = (x << 8) + (*result)[i];
    (*result)[i] = x >> 1;
    x &= 1;
}
```

单行注释

```
DoSomething();           // Comment here so the comments line up.
DoSomethingElseThatIsLonger(); // Comment here so there are two spaces between
                             // the code and the comment.

{
    DoSomethingElse();    // One space before line comments normally.
}
```

参数注释

```
bool success = CalculateSomething(interesting_value,
                                10,      // Default base value.
                                false,   // Not the first time we're calling this.
                                NULL);   // No callback.
```

格式规范

行长

保证在不滚动滚动条的情况下能阅读完一行代码，否则进行断行。

空格和退格

设置你的编辑器将退格键更换为 4 个空格键。

函数声明和定义

尽量保证函数返回值、函数名和参数同在一行，如果太长，可适当换行。如下所示：

```
ReturnType ClassName::FunctionName(Type par_name1, Type par_name2) {
    DoSomething();
    ...
}

ReturnType ClassName::ReallyLongFunctionName(Type par_name1, Type par_name2,
                                              Type par_name3) {
    DoSomething();
    ...
}
```

函数调用

如果不能完全显示在一行，可考虑断行。使用样例如下所示：

```
bool retval = DoSomething(argument1, argument2, argument3);

bool retval = DoSomething(averyveryveryverylongargument1,
                          argument2, argument3);

bool retval = DoSomething(argument1,
                          argument2,
                          argument3,
                          argument4);
```

条件语句

所有条件语句必须用大括号进行格式化，即使只有一条语句。使用样例如下所示：

```
if (condition) {
    ...
}
```



```
}  
else if (...) {  
    ...  
}  
else {  
    ...  
}
```

循环和选择语句

代码体必须用大括号进行格式化，即使是空代码体。使用样例如下：

```
switch (var) {  
    case 0: { // first case  
        ...  
        break;  
    }  
    case 1: {  
        ...  
        break;  
    }  
    default: {  
        assert(false);  
    }  
}  
  
while (condition) {  
    // Repeat test until it returns false.  
}  
  
for (int i = 0; i < some_number; ++i) {  
    ...  
}
```

指针和引用

指针符和引用符与变量之间没有空格。如下所示：

```
x = *p;  
p = &x;  
x = r.y;  
x = r->y;  
  
char *c, *d, *e;  
const string &str;  
  
char* c, * d; // not allowed
```